Are we fully leveraging TableGen in MLIR?

KSHITIJ JAIN

COMPILER ENGINEER AT D-MATRIX

Goal

- Demonstrate that richer TableGen code can –
  - Reduce the mental overhead on compiler developers.
  - Make a compiler's behavior more apparent and robust.
  - Lower the barrier to entry for new contributors.

- Show—via concrete examples—how the above mentioned benefits can be realized using existing concepts/utilities, and often overlooked software engineering pragmatisms.

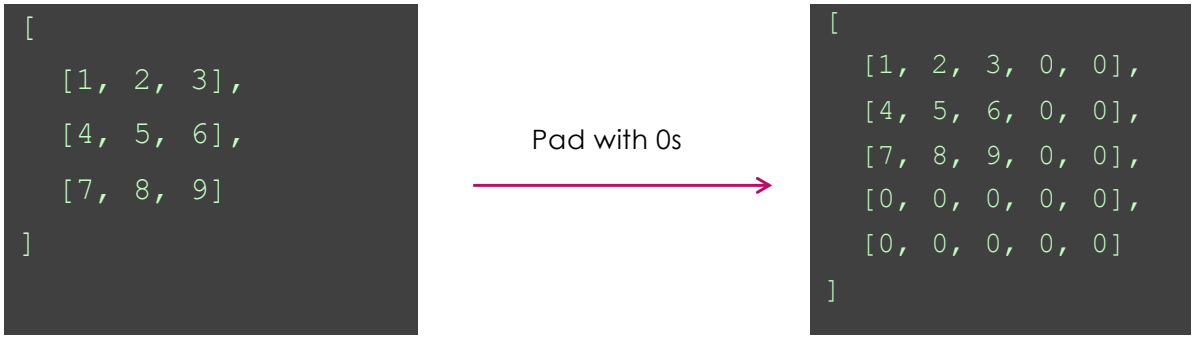
Note – Some familiarity with TableGen and its usage in MLIR is expected.

Reduce the mental overhead on compiler developers

Imagine you worked on a machine learning compiler and were tasked with creating a Pad op.

Requirements –

1. Pads inner 2 dimensions of an N-dimensional row-major tensor, where N >= 2.
2. Performs edge padding - bottom and right edges extended by a specified padding value.



3x3xf32

5x5xf32

Bad TableGen Code

```
def PadOp : Op<"padOp", [Pure]> {
  let arguments = (ins
          AnyRankedTensor:$input,
          AnyAttr:$pad_value
  );

  let results = (outs
          AnyRankedTensor:$output
  );
}
```

Bad TableGen Code

```
def PadOp : Op<"padOp", [Pure]> {
  let arguments = (ins
         AnyRankedTensor:$input,
         AnyAttr:$pad_value
  );

  let results = (outs
         AnyRankedTensor:$output
  );
}
```

The invariants -

Bad TableGen Code

```
def PadOp : Op<"padOp", [Pure]> {
  let arguments = (ins
         AnyRankedTensor:$input,
         AnyAttr:$pad_value
  );

  let results = (outs
         AnyRankedTensor:$output
  );
}
```

The invariants -

1. **Input and output tensor ranks MUST be the same.**

Bad TableGen Code

```
def PadOp : Op<"padOp", [Pure]> {
  let arguments = (ins
         AnyRankedTensor:$input,
         AnyAttr:$pad_value
  );

  let results = (outs
         AnyRankedTensor:$output
  );
}
```

The invariants -

1. Input and output tensor ranks MUST be the same.
2. **Input, output, and pad_value element types MUST be the same.**

Bad TableGen Code

```
def PadOp : Op<"padOp", [Pure]> {
  let arguments = (ins
         AnyRankedTensor:$input,
         AnyAttr:$pad_value
  );

  let results = (outs
         AnyRankedTensor:$output
  );
}
```

The invariants -

1. Input and output tensor ranks MUST be the same.
2. Input, output, and pad_value element types MUST be the same.
3. **Input and output dimension sizes MUST be the same, barring the 2 innermost dimensions.**

Bad TableGen Code

```
def PadOp : Op<"padOp", [Pure]> {
  let arguments = (ins
         AnyRankedTensor:$input,
         AnyAttr:$pad_value
  );

  let results = (outs
         AnyRankedTensor:$output
  );
}
```

The invariants -

1. Input and output tensor ranks MUST be the same.
2. Input, output, and pad_value element types MUST be the same.
3. Input and output dimension sizes MUST be the same, barring the 2 innermost dimensions.

Without guarantees on invariants –

Issues due to bad TableGen Code

```
def PadOp : Op<"padOp", [Pure]> {
  let arguments = (ins
         AnyRankedTensor:$input,
         AnyAttr:$pad_value
  );

  let results = (outs
         AnyRankedTensor:$output
  );
}
```

The invariants -

1. Input and output tensor ranks will be the same.
2. Input, output, and pad_value element types will be the same.
3. Input and output dimension sizes will be the same, barring the 2 innermost dimensions.

Without guarantees on invariants –

1. **Semantic and syntactic gap widens.**

Issues due to bad TableGen Code

```
def PadOp : Op<"padOp", [Pure]> {
  let arguments = (ins
         AnyRankedTensor:$input,
         AnyAttr:$pad_value
  );

  let results = (outs
         AnyRankedTensor:$output
  );
}
```

The invariants -

1. Input and output tensor ranks will be the same.
2. Input, output, and pad_value element types will be the same.
3. Input and output dimension sizes will be the same, barring the 2 innermost dimensions.

Without guarantees on invariants –

1. Semantic and syntactic gap widens.
2. **Lowering process gets complicated (and often sprinkled with asserts).**

Issues due to bad TableGen Code

```
def PadOp : Op<"padOp", [Pure]> {
  let arguments = (ins
         AnyRankedTensor:$input,
         AnyAttr:$pad_value
  );

  let results = (outs
         AnyRankedTensor:$output
  );
}
```

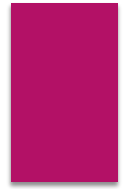
The invariants -

1. Input and output tensor ranks will be the same.
2. Input, output, and pad_value element types will be the same.
3. Input and output dimension sizes will be the same, barring the 2 innermost dimensions.

Without guarantees on invariants –

1. Semantic and syntactic gap widens.
2. Lowering process gets complicated (and often sprinkled with asserts).
3. **Hand writing IR becomes (even more) error prone.**

Good TableGen Code



```
def EdgePadOp : Op<”edge_pad", [
  Pure,
  AllRanksMatch<["input", "output"]>
  AllElementTypesMatch<["input", "output”, “pad_value]>,
]> {
  let arguments = (ins
    RankedTensorOf<[I32, F32]>:$input,
    TypedAttrInterface:$pad_value
  );

  let results = (outs
    RankedTensorOf<[I32, F32]>:$output
  );
}
```

Good TableGen Code

```
def EdgePadOp : Op<”edge_pad", [
  Pure,
  AllRanksMatch<["input", "output"]>
  AllElementTypesMatch<["input", "output”, “pad_value]>,
]> {
  let arguments = (ins
    RankedTensorOf<[I32, F32]>:$input,
    TypedAttrInterface:$pad_value
  );

  let results = (outs
    RankedTensorOf<[I32, F32]>:$output
  );
}
```

The invariants -

Good TableGen Code

```
def EdgePadOp : Op<”edge_pad", [
  Pure,
  AllRanksMatch<["input", "output"]>
  AllElementTypesMatch<["input", "output”, “pad_value]>,
]> {
  let arguments = (ins
    RankedTensorOf<[I32, F32]>:$input,
    TypedAttrInterface:$pad_value
  );

  let results = (outs
    RankedTensorOf<[I32, F32]>:$output
  );
}
```

The invariants -

1. **Input and output tensor ranks will be the same.**

Good TableGen Code

```
def EdgePadOp : Op<”edge_pad", [
  Pure,
  AllRanksMatch<["input", "output"]>
  AllElementTypesMatch<["input", "output”, "pad_value]>,
]> {
  let arguments = (ins
    RankedTensorOf<[I32, F32]>:$input,
    TypedAttrInterface:$pad_value
  );

  let results = (outs
    RankedTensorOf<[I32, F32]>:$output
  );
}
```

The invariants -

1. Input and output tensor ranks will be the same.
2. **Input, output, and pad_value element types will be the same.**

Good TableGen Code

```
def EdgePadOp : Op<"edge_pad", [
  Pure,
  AllRanksMatch<["input", "output"]>
  AllElementTypesMatch<["input", "output", "pad_value]>,
]> {
  let arguments = (ins
    RankedTensorOf<[I32, F32]>:$input,
    TypedAttrInterface:$pad_value
  );

  let results = (outs
    RankedTensorOf<[I32, F32]>:$output
  );

  // Invariant 3 can be enforced directly in TableGen,
  // but when things get more complicated, it might be
  // best to move to cpp verifiers.
  //
  // They provide the exact same benefits, although you do
  // some information locality.
  let hasVerifier = 1;
}
```

The invariants -

1. Input and output tensor ranks will be the same.
2. Input, output, and pad_value element types will be the same.
3. **Input and output dimension sizes will be the same, barring the 2 innermost dimensions.**

Good TableGen Code

```
def EdgePadOp : Op<"edge_pad", [
  Pure,
  AllRanksMatch<["input", "output"]>
  AllElementTypesMatch<["input", "output", "pad_value]>,
]> {
  let arguments = (ins
    RankedTensorOf<[I32, F32]>:$input,
    TypedAttrInterface:$pad_value
  );

  let results = (outs
    RankedTensorOf<[I32, F32]>:$output
  );

  // Invariant 3 can be enforced directly in TableGen,
  // but when things get more complicated, it might be
  // best to move to cpp verifiers.
  //
  // They provide the exact same benefits, although you do
  // some information locality.
  let hasVerifier = 1;
}
```

The invariants -

1. Input and output tensor ranks will be the same.
2. Input, output, and pad_value element types will be the same.
3. **Input and output dimension sizes will be the same, barring the 2 innermost dimensions.**

With guarantees on invariants –

Good TableGen Code

```
def EdgePadOp : Op<”edge_pad", [
  Pure,
  AllRanksMatch<["input", "output"]>
  AllElementTypesMatch<["input", "output”, “pad_value]>,
]> {
  let arguments = (ins
    RankedTensorOf<[I32, F32]>:$input,
    TypedAttrInterface:$pad_value
  );

  let results = (outs
    RankedTensorOf<[I32, F32]>:$output
  );


  // Invariant 3 can be enforced directly in TableGen,
  // but when things get more complicated, it might be
  // best to move to cpp verifiers.
  //
  // They provide the exact same benefits, although you do
  // some information locality.
  let hasVerifier = 1;
}
```

The invariants -

1. Input and output tensor ranks will be the same.
2. Input, output, and pad_value element types will be the same.
3. **Input and output dimension sizes will be the same, barring the 2 innermost dimensions.**

With guarantees on invariants –

1. **Correctness by construction.**

Good TableGen Code

```
def EdgePadOp : Op<"edge_pad", [
  Pure,
  AllRanksMatch<["input", "output"]>
  AllElementTypesMatch<["input", "output", "pad_value]>,
]> {
  let arguments = (ins
    RankedTensorOf<[I32, F32]>:$input,
    TypedAttrInterface:$pad_value
  );

  let results = (outs
    RankedTensorOf<[I32, F32]>:$output
  );

  // Invariant 3 can be enforced directly in TableGen,
  // but when things get more complicated, it might be
  // best to move to cpp verifiers.
  //
  // They provide the exact same benefits, although you do
  // some information locality.
  let hasVerifier = 1;
}
```

The invariants -

1. Input and output tensor ranks will be the same.
2. Input, output, and pad_value element types will be the same.
3. **Input and output dimension sizes will be the same, barring the 2 innermost dimensions.**

With guarantees on invariants –

1. Correctness by construction.
2. **Lowering process stays mechanical**

Good TableGen Code

```
def EdgePadOp : Op<"edge_pad", [
  Pure,
  AllRanksMatch<["input", "output"]>
  AllElementTypesMatch<["input", "output", "pad_value]>,
]> {
  let arguments = (ins
    RankedTensorOf<[I32, F32]>:$input,
    TypedAttrInterface:$pad_value
  );

  let results = (outs
    RankedTensorOf<[I32, F32]>:$output
  );

  // Invariant 3 can be enforced directly in TableGen,
  // but when things get more complicated, it might be
  // best to move to cpp verifiers.
  //
  // They provide the exact same benefits, although you do
  // some information locality.
  let hasVerifier = 1;
}
```

The invariants -

1. Input and output tensor ranks will be the same.
2. Input, output, and pad_value element types will be the same.
3. **Input and output dimension sizes will be the same, barring the 2 innermost dimensions.**

With guarantees on invariants –

1. Correctness by construction.
2. Lowering process stays mechanical
3. **Hand writing IR becomes becomes easier to get write and tools such as MLIR's LSP server can be of assistance.**

Takeaways
for
reducing the mental overhead on compiler developers

Takeaways
for
reducing the mental overhead on compiler developers

Specify op invariants upfront as much as possible to…

Takeaways for reducing the mental overhead on compiler developers

Specify op invariants upfront as much as possible to…

1. **Get correctness by construction guarantees and reduce the gap b/w the syntax and semantics of your op.**

Takeaways for reducing the mental overhead on compiler developers

Specify op invariants upfront as much as possible to…

1. Get correctness by construction guarantees and reduce the gap b/w the syntax and semantics of your op.
2. **Make op lowering process as mechanical as possible.**

Takeaways for reducing the mental overhead on compiler developers

Specify op invariants upfront as much as possible to…

1. Get correctness by construction guarantees and reduce the gap b/w the syntax and semantics of your op.
2. Make op lowering process as mechanical as possible.
3. **Get better assistance from and integration w/ development tools such as, MLIR's LSP server.**

Takeaways for reducing the mental overhead on compiler developers

Specify op invariants upfront as much as possible to…

1. Get correctness by construction guarantees and reduce the gap b/w the syntax and semantics of your op.
2. Make op lowering process as mechanical as possible.
3. Get better assistance from and integration w/ development tools such as, MLIR's LSP server.

Everything applies to attributes and types as well!

Takeaways
for
reducing the mental overhead on compiler developers

Op constraints – common upstream op constraints can be found here.

Attribute constraints – common upstream attribute constraints can be found here.

Type constraints – common upstream type constraints can be found here.

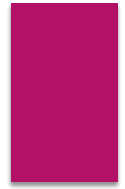
Make a compiler's behavior more apparent and robust

Imagine you worked on a machine learning compiler and were tasked with designing a Reshape op.

Requirements –

1. Reshapes an input tensor.
2. Does not change input tensor's data and number of elements.

Bad TableGen Code



```
def ReshapeOp : Op<”reshape", [Pure]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );
}
```

Issues due to bad TableGen Code

```
def ReshapeOp : Op<"reshape", [Pure]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );
}
```

The invariants -

Issues due to bad TableGen Code

```
def ReshapeOp : Op<"reshape", [Pure]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );
}
```

The invariants -

1. **input and output element types MUST be the same.**

Issues due to bad TableGen Code

```
def ReshapeOp : Op<"reshape", [Pure]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );
}
```

The invariants -

1. input and output element types MUST be the same.
2. **input and output element counts MUST be the same.**

Issues due to bad TableGen Code

```
def ReshapeOp : Op<"reshape", [Pure]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );
}
```

The invariants -

1. input and output element types MUST be the same.
2. input and output element counts MUST be the same.
3. **new_shape attribute MUST match the output's shape.**

Issues due to bad TableGen Code

```
def ReshapeOp : Op<"reshape", [Pure]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );
}
```

The invariants -

1. input and output element types MUST be the same.
2. input and output element counts MUST be the same.
3. new_shape attribute MUST match the output's shape.

Without guarantees on invariants –

Issues due to bad TableGen Code

```
def ReshapeOp : Op<"reshape", [Pure]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );
}
```

The invariants -

1. input and output element types MUST be the same.
2. input and output element counts MUST be the same.
3. new_shape attribute MUST match the output's shape.

Without guarantees on invariants –

1. **Non-sensical op.**

Issues due to bad TableGen Code

```
def ReshapeOp : Op<"reshape", [Pure]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );
}
```

The invariants -

1. input and output element types MUST be the same.
2. input and output element counts MUST be the same.
3. new_shape attribute MUST match the output's shape.

Without guarantees on invariants –

1. Non-sensical op.
2. **Confusing error cascade.**

You might have a reshape op like this -

```
%1 = reshape %0 {new_shape = [2, 4, 64]} : tensor<4x128xf32> -> tensor<2x4x32xf32>
```

You might have a reshape op like this -

```
%1 = reshape %0 {new_shape = [2, 4, 64]} : tensor<4x128xf32> -> tensor<2x4x32xf32>
```

You might have a reshape op like this -

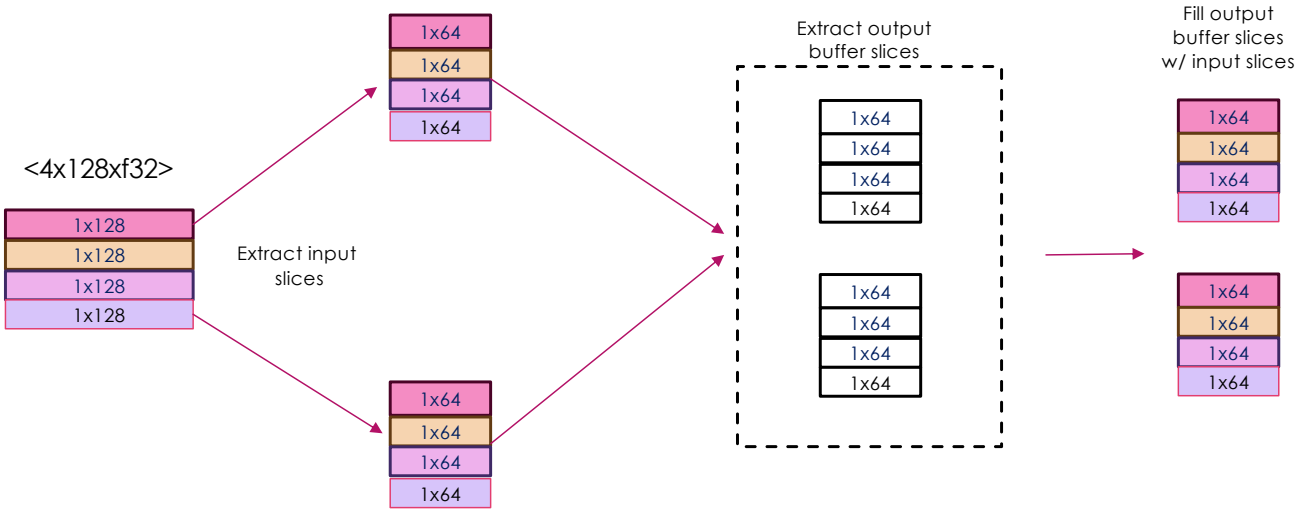
```
%1 = reshape %0 {new_shape = [2, 4, 64]} : tensor<4x128xf32> -> tensor<2x4x32xf32>
```

You might have a reshape op like this -

```
%1 = reshape %0 {new_shape = [2, 4, 64]} : tensor<4x128xf32> -> tensor<2x4x32xf32>
```

You might have a reshape op like this -

```
%1 = reshape %0 {new_shape = [2, 4, 64]} : tensor<4x128xf32> -> tensor<2x4x32xf32>
```



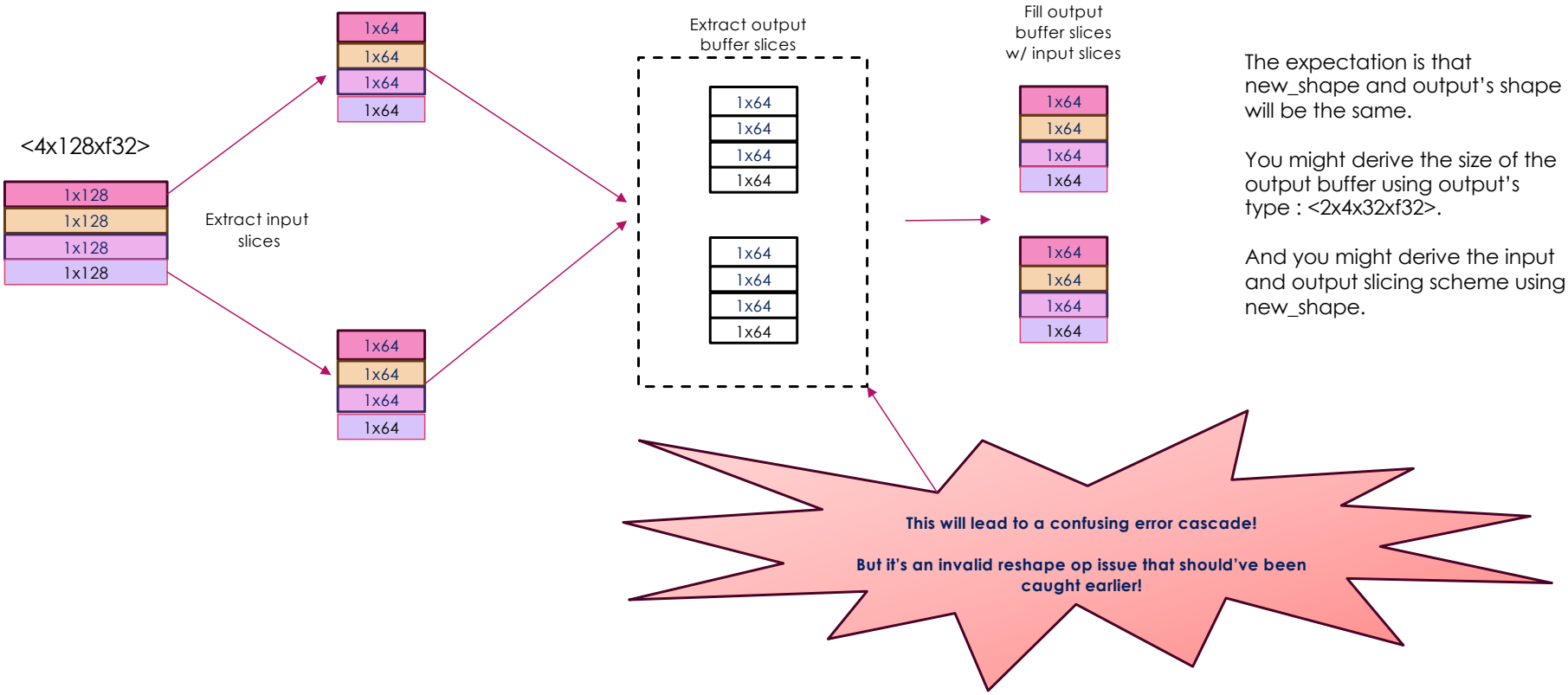
The expectation is that new_shape and output's shape will be the same.

You might derive the size of the output buffer using output's type : <2x4x32xf32>.

And you might derive the input and output slicing scheme using new_shape.

You might have a reshape op like this -

```
%1 = reshape %0 {new_shape = [2, 4, 64]} : tensor<4x128xf32> -> tensor<2x4x32xf32>
```

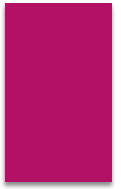


The expectation is that new_shape and output's shape will be the same.

You might derive the size of the output buffer using output's type : <2x4x32xf32>.

And you might derive the input and output slicing scheme using new_shape.

Good op definition



```
def ReshapeOp : Op<"reshape", [
  Pure,
  AllElementTypesMatch<"input", "output">,
  AllElementCountsMatch<"input", "output">,
]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );

  let hasVerifier = 1;
}
```

Good op definition

```
def ReshapeOp : Op<"reshape", [
  Pure,
  AllElementTypesMatch<"input", "output">,
  AllElementCountsMatch<"input", "output">,
]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );

  let hasVerifier = 1;
}
```

The invariants -

Good op definition

```
def ReshapeOp : Op<"reshape", [
  Pure,
  AllElementTypesMatch<"input", "output">,
  AllElementCountsMatch<"input", "output">,
]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );

  let hasVerifier = 1;
}
```

The invariants -

1. **input and output element types will be the same.**

Good op definition

```
def ReshapeOp : Op<"reshape", [
  Pure,
  AllElementTypesMatch<"input", "output">,
  AllElementCountsMatch<"input", "output">,
]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );

  let hasVerifier = 1;
}
```

The invariants -

1. input and output element types will be the same.
2. **input and output element counts will be the same.**

Good op definition

```
def ReshapeOp : Op<"reshape", [
  Pure,
  AllElementTypesMatch<"input", "output">,
  AllElementCountsMatch<"input", "output">,
]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );

  let hasVerifier = 1;
}
```

The invariants -

1. input and output element types will be the same.
2. input and output element counts will be the same.
3. **new_shape attribute will match the output's shape.**

Good op definition

```
def ReshapeOp : Op<"reshape", [
  Pure,
  AllElementTypesMatch<"input", "output">,
  AllElementCountsMatch<"input", "output">,
]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );

  let hasVerifier = 1;
}
```

The invariants -

1. input and output element types will be the same.
2. input and output element counts will be the same.
3. new_shape attribute must match the output's shape.

With guarantees on invariants –

Good op definition

```
def ReshapeOp : Op<"reshape", [
  Pure,
  AllElementTypesMatch<"input", "output">,
  AllElementCountsMatch<"input", "output">,
]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );

  let hasVerifier = 1;
}
```

The invariants -

1. input and output element types will be the same.
2. input and output element counts will be the same.
3. new_shape attribute must match the output's shape.

With guarantees on invariants –

1. **Sensible op**

Good op definition

```
def ReshapeOp : Op<"reshape", [
  Pure,
  AllElementTypesMatch<"input", "output">,
  AllElementCountsMatch<"input", "output">,
]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );

  let hasVerifier = 1;
}
```

The invariants -

1. input and output element types will be the same.
2. input and output element counts will be the same.
3. new_shape attribute must match the output's shape.

With guarantees on invariants –

1. Sensible op
2. **No confusing error cascades**

Good op definition

```
def ReshapeOp : Op<"reshape", [
  Pure,
  AllElementTypesMatch<"input", "output">,
  AllElementCountsMatch<"input", "output">,
]> {
  let arguments = (ins
    AnyRankedTensor : $input,
    I64ArrayAttr : $new_shape
  );

  let results = (outs
    AnyRankedTensor : $output,
  );

  let hasVerifier = 1;
}
```

new_shape is not strictly needed.

Can be derived from output.

Best to deduplicate redundant information from your ops

Takeaways
for
making a compiler's behavior more apparent and robust

Takeaways
for
making a compiler's behavior more apparent and robust

Specify op invariants upfront as much as possible and deduplicate redundant information from your op definitions to…

Takeaways for making a compiler's behavior more apparent and robust

Specify op invariants upfront as much as possible and deduplicate redundant information from your op definitions to…

1. Avoid confusing error cascades and enable early failures where context is apparent.

Takeaways for making a compiler's behavior more apparent and robust

Specify op invariants upfront as much as possible and deduplicate redundant information from your op definitions to…

1. Avoid confusing error cascades and enable early failures where context is apparent.
2. **Avoid having multiple data sources for the same information.**

Lower the barrier to entry for new contributors

Imagine you're starting out as a compiler engineer, at a company that owns a machine learning compiler and your first task is to add support for a 2D convolution op to the compiler FE.

Bad TableGen Code

```
def ConvOp : Op<"conv”> {
  let summary = "Convolution operator";

  let description = [{
    Applies a convolution over an input signal composed of several input
    planes.
  }];

  let arguments = (ins
    AnyTensor : $input,
    AnyTensor : $weight,
    AnyTensor : $bias,
    DenseI64ArrayAttr : $pad,
    DenseI64ArrayAttr : $stride,
    DenseI64ArrayAttr : $dilation
  );

  let results = (
    outs AnyTensor : $output
  );
}
```

Issues with bad TableGen Code

```
def ConvOp : Op<"conv”> {
  let summary = "Convolution operator";

  let description = [{
    Applies a convolution over an input signal composed of several input
    planes.
  }];

  let arguments = (ins
    AnyTensor : $input,
    AnyTensor : $weight,
    AnyTensor : $bias,
    DenseI64ArrayAttr : $pad,
    DenseI64ArrayAttr : $stride,
    DenseI64ArrayAttr : $dilation
  );

  let results = (
    outs AnyTensor : $output
  );
}
```

1. **What is convolution?**

Issues with bad TableGen Code

```
def ConvOp : Op<"conv”> {
  let summary = "Convolution operator";

  let description = [{
    Applies a convolution over an input signal composed of several input
    planes.
  }];

  let arguments = (ins
    AnyTensor : $input,
    AnyTensor : $weight,
    AnyTensor : $bias,
    DenseI64ArrayAttr : $pad,
    DenseI64ArrayAttr : $stride,
    DenseI64ArrayAttr : $dilation
  );

  let results = (
    outs AnyTensor : $output
  );
}
```

1. What is convolution?
2. **Do I have a real task?**

Issues with bad TableGen Code

```
def ConvOp : Op<"conv”> {
  let summary = "Convolution operator";

  let description = [{
    Applies a convolution over an input signal composed of several input
    planes.
  }];

  let arguments = (ins
    AnyTensor : $input,
    AnyTensor : $weight,
    AnyTensor : $bias,
    DenseI64ArrayAttr : $pad,
    DenseI64ArrayAttr : $stride,
    DenseI64ArrayAttr : $dilation
  );

  let results = (
    outs AnyTensor : $output
  );
}
```

1. What is convolution?
2. Do I have a real task?
3. **Perhaps this is a 1D convolution op?**

Issues with bad TableGen Code

```
def ConvOp : Op<"conv”> {
  let summary = "Convolution operator";

  let description = [{
    Applies a convolution over an input signal composed of several input
    planes.
  }];

  let arguments = (ins
    AnyTensor : $input,
    AnyTensor : $weight,
    AnyTensor : $bias,
    DenseI64ArrayAttr : $pad,
    DenseI64ArrayAttr : $stride,
    DenseI64ArrayAttr : $dilation
  );

  let results = (
    outs AnyTensor : $output
  );
}
```

1. What is convolution?
2. Do I have a real task?
3. Perhaps this is a 1D convolution op?
4. **Can I reuse this for 2D convolutions?**

Good TableGen Code

```
def Conv1DOp : Op<"conv1d", [
  Pure,
  AllElementTypesMatch<["input", "weight", "bias", "output"]>
]> {
  let summary = ...;
  let description = ...;
  let arguments = (ins
    3DTensorOf<[f32]> : $input,
    3DTensorOf<[f32]> : $weight,
    1DTensorOf<[f32]> : $bias,
    I64Attr : $pad,
    I64Attr : $stride,
    I64Attr : $dilation
  );
  let results = (outs
    3DTensorOf<[f32]> : $output
  );
 let hasVerifier = 1;
}
```

Good TableGen Code

```
def Conv1DOp : Op<"conv1d", [

  Pure,

  AllElementTypesMatch<["input", "weight", "bias", "output"]>

]> {

  let description = [{

    A convolution is a sliding window operation that applies a filter across an input
    sequence, computing weighted sums at each position to extract features.

    This op applies a 1D convolution over an input signal. Input signal is assumned to
    be of the format of NCL where "N" (batch size), "L" (length), and "C" (channels).

    This assumption is made to be consistent with the PyTorch only.

    Arguments:
    `input` is a 3D tensor of shape [N, Cin, Lin].
    `weight` is a 3D tensor of shape [Cout, Cin, K] where K is the kernel length.
    `bias` is a 1D tensor of shape [Cout].

    Results:
    `output` is a 3D tensor of shape [N, Cout, Lout].

    Attributes:

   `pad` is a scalar the defines the amount of padding applied on both sides.
   `stride` is a scalar that controls the stride length of the sliding-window in
   the conv1d.
   `dilation` is a scalar that controls the spacing between the kernel points.

  }];

  ...

}
```

Good TableGen Code

```
def Conv1DOp : Op<"conv1d", [

  Pure,

  AllElementTypesMatch<["input", "weight", "bias", "output"]>

]> {

  let description = [{

    A convolution is a sliding window operation that applies a filter across an input
    sequence, computing weighted sums at each position to extract features.

    This op applies a 1D convolution over an input signal. Input signal is assumned to
    be of the format of NCL where "N" (batch size), "L" (length), and "C" (channels).

    This assumption is made to be consistent with the PyTorch only.

    Arguments:
    `input` is a 3D tensor of shape [N, Cin, Lin].
    `weight` is a 3D tensor of shape [Cout, Cin, K] where K is the kernel length.
    `bias` is a 1D tensor of shape [Cout].

    Results:
    `output` is a 3D tensor of shape [N, Cout, Lout].

    Attributes:

   `pad` is a scalar the defines the amount of padding applied on both sides.
   `stride` is a scalar that controls the stride length of the sliding-window in
   the conv1d.
   `dilation` is a scalar that controls the spacing between the kernel points.

  }];

  ...

}
```

Good TableGen Code

```
def Conv1DOp : Op<"conv1d", [

  Pure,

  AllElementTypesMatch<["input", "weight", "bias", "output"]>

]> {

  let description = [{

    A convolution is a sliding window operation that applies a filter across an input
    sequence, computing weighted sums at each position to extract features.

    This op applies a 1D convolution over an input signal. Input signal is assumned to
    be of the format of NCL where "N" (batch size), "L" (length), and "C" (channels).

    This assumption is made to be consistent with the PyTorch only.

    Arguments:
    `input` is a 3D tensor of shape [N, Cin, Lin].
    `weight` is a 3D tensor of shape [Cout, Cin, K] where K is the kernel length.
    `bias` is a 1D tensor of shape [Cout].

    Results:
    `output` is a 3D tensor of shape [N, Cout, Lout].

    Attributes:

    `pad` is a scalar the defines the amount of padding applied on both sides.
    `stride` is a scalar that controls the stride length of the sliding-window in
    the conv1d.
    `dilation` is a scalar that controls the spacing between the kernel points.

  }];

  ...

}
```

Good TableGen Code

```
def Conv1DOp : Op<"conv1d", [
  Pure,
  AllElementTypesMatch<["input", "weight", "bias", "output"]>
]> {
  let summary = ...;
  let description = ...;
  let arguments = (ins
    3DTensorOf<[f32]> : $input,
    3DTensorOf<[f32]> : $weight,
    1DTensorOf<[f32]> : $bias,
    I64Attr : $pad,
    I64Attr : $stride,
    I64Attr : $dilation
  );
  let results = (outs
    3DTensorOf<[f32]> : $output
  );
 let hasVerifier = 1;
}
```

Good TableGen Code

```
def Conv1DOp : Op<"conv1d", [
  Pure,
  AllElementTypesMatch<["input", "weight", "bias", "output"]>
]> {
  let summary = ...;
  let description = ...;
  let arguments = (ins
    3DTensorOf<[f32]> : $input,
    3DTensorOf<[f32]> : $weight,
    1DTensorOf<[f32]> : $bias,
    I64Attr : $pad,
    I64Attr : $stride,
    I64Attr : $dilation
  );
  let results = (outs
    3DTensorOf<[f32]> : $output
  );
 let hasVerifier = 1;
}
```

1. **What is convolution?**
2. Do I have a real task?
3. Perhaps this is a 1D convolution op?
4. Can I reuse this for 2D convolutions?

Good TableGen Code

```
def Conv1DOp : Op<"conv1d", [
  Pure,
  AllElementTypesMatch<["input", "weight", "bias", "output"]>
]> {
  let summary = ...;
  let description = ...;
  let arguments = (ins
    3DTensorOf<[f32]> : $input,
    3DTensorOf<[f32]> : $weight,
    1DTensorOf<[f32]> : $bias,
    I64Attr : $pad,
    I64Attr : $stride,
    I64Attr : $dilation
  );
  let results = (outs
    3DTensorOf<[f32]> : $output
  );
 let hasVerifier = 1;
}
```

1. What is convolution?
2. **Do I have a real task?**
3. **Perhaps this is a 1D convolution op?**
4. Can I reuse this for 2D convolutions?

Good TableGen Code

```
def Conv1DOp : Op<"conv1d", [
  Pure,
  AllElementTypesMatch<["input", "weight", "bias", "output"]>
]> {
  let summary = ...;
  let description = ...;
  let arguments = (ins
    3DTensorOf<[f32]> : $input,
    3DTensorOf<[f32]> : $weight,
    1DTensorOf<[f32]> : $bias,
    I64Attr : $pad,
    I64Attr : $stride,
    I64Attr : $dilation
  );
  let results = (outs
    3DTensorOf<[f32]> : $output
  );
 let hasVerifier = 1;
}
```

1. What is convolution?
2. Do I have a real task?
3. Perhaps this is a 1D convolution op?
4. **Can I reuse this for 2D convolutions?**

Good TableGen Code

```
def Conv1DOp : Op<"conv1d", [
  Pure,
  AllElementTypesMatch<["input", "weight", "bias", "output"]>
]> {
  let summary = ...;
  let description = ...;
  let arguments = (ins
    3DTensorOf<[f32]> : $input,
    3DTensorOf<[f32]> : $weight,
    1DTensorOf<[f32]> : $bias,
    I64Attr : $pad,
    I64Attr : $stride,
    I64Attr : $dilation
  );
  let results = (outs
    3DTensorOf<[f32]> : $output
  );
 let hasVerifier = 1;
}
```

The op definition clearly indicates the feature supported by the compiler.

Takeaways
for
lowering the barrier to entry for new contributors

Takeaways
for
lowering the barrier to entry for new contributors

Specify op invariants upfront as much as possible and write elaborate op descriptions to…

Takeaways for lowering the barrier to entry for new contributors

Specify op invariants upfront as much as possible and write elaborate op descriptions to…

1. **Localize all the information new contributors might need to work with the op.**

Takeaways for lowering the barrier to entry for new contributors

Specify op invariants upfront as much as possible and write elaborate op descriptions to…

1. Localize all the information new contributors might need to work with the op.
2. **Clearly indicate the compiler's feature set to new contributors.**

Takeaways for lowering the barrier to entry for new contributors

Specify op invariants upfront as much as possible and write elaborate op descriptions to…

1. Localize all the information new contributors might need to work with the op.
2. Clearly indicate the compiler's feature set to new contributors.

Thank you!

Chime in with your opinions on the following LLVM discourse -

https://discourse.llvm.org/t/survey-interested-in-discussing-richer-tablegen-descriptions-in-mlir/87959